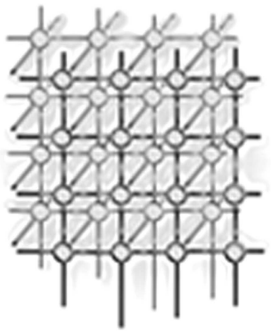
NIC-assisted cache-efficient receive stack for message passing over Ethernet

Brice Goglin

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2010; :1–15 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*



NIC-assisted cache-efficient receive stack for message passing over Ethernet

Brice Goglin

INRIA Bordeaux - Sud-Ouest
351, cours de la Libération
F-33405 Talence cedex – France

SUMMARY

High-speed networking in clusters usually relies on advanced hardware features in the NICs, such as zero-copy capability. Open-MX is a high-performance message passing stack tailored for regular Ethernet hardware without such capabilities. We present the addition of a multiqueue support in the Open-MX receive stack so that all incoming packets for the same process are handled on the same core. We then introduce the idea of binding the target end process near its dedicated receive queue. This model leads to a more cache-efficient receive stack for Open-MX. It also proves that very simple and stateless hardware features may have a significant impact on message passing performance over Ethernet. The implementation of this model in a firmware reveals that it may not be as efficient as some manually tuned micro-benchmarks. But our multiqueue receive stack generally performs better than the original single queue stack, especially on large communication patterns where multiple processes are involved and manual binding is difficult.

KEY WORDS: High-speed networking; Cache; MPI; Ethernet; Multiqueue; Binding

1. INTRODUCTION

The emergence of 10-gigabit/s ETHERNET hardware raised the questions of when and how the long-awaited convergence with high-speed networks will become a reality. ETHERNET now appears as an interesting networking layer within local area networks for various protocols such as FCoE [7]. Meanwhile, several network vendors that previously focused on high-performance computing added interoperability with ETHERNET to their hardware, such as MELLANOX CONNECTX [6] or MYRICOM MYRI-10G [17]. However, these technologies still require dedicated interfaces in the nodes. The gap between these advanced NICs and regular ETHERNET NICs remains substantial. It brings the question of which hardware feature will become legacy once the actual convergence will be reached.

Several research works were carried out in the context of high-performance message passing over ETHERNET as a way to improve the overall parallel computing performance without requiring expensive networking hardware. GAMMA [5] or EMP [22] only work on a limited spectrum of hardware since they use modified drivers or hardware. Our OPEN-MX [10] stack is another message passing model implemented on top of the ETHERNET software layer of the LINUX kernel. It offers high-performance communication over any generic ETHERNET hardware using the wire specifications and the application programming interface of *Myrinet Express* [18]. However, like QMP [3] and PM [23] (or any other software-based message passing), being compatible with any legacy ETHERNET NICs also means that OPEN-MX suffers from limited hardware features. For instance, it has to work around the inability to perform zero-copy receive by offloading memory copy on INTEL *I/O Acceleration Technology* (I/OAT) [11].

We propose to improve the cache-efficiency of the receive side of ETHERNET-based message passing by extending the hardware IP multiqueue support to filter OPEN-MX packets as well. Such a stateless feature requires very little computing power and software support compared to the existing complex and stateful features such as zero-copy or TOE (*TCP Offload Engine*). Parallelizing the stack is known to be important on modern machines [24]. We are looking at it in the context of binding the whole packet processing to the same core, from the bottom interrupt handler up to the application.

This paper is an extended revision of [12] organized as follows. We present OPEN-MX, its possible cache-inefficiency problems, related works and our motivations in Section 2. Section 3 describes our proposal to combine the multiqueue extension in the MYRI-10G firmware and its corresponding support in OPEN-MX so as to we build an automatic binding facility for both the receive handler in the driver and the target application. Section 4 presents a performance evaluation which shows that our model achieves satisfying performance for micro-benchmarks, reduces the overall cache miss rate, and improves performance in the case of large communication patterns.

2. BACKGROUND AND MOTIVATIONS

In this section, we briefly describe the OPEN-MX stack and how the cache is involved on the receive side. We then present previous works on the cache efficiency of high-performance networking stacks. We finally detail our motivation to add some OPEN-MX specific support in the NIC and our objectives with this implementation.

2.1. Cache Efficiency Issues in the Open-MX Stack

The OPEN-MX stack* aims at providing high-performance message passing over any generic ETHERNET hardware. First, it bypasses the usual TCP/IP stack so as to exploit the networking

*Available for download at `http://open-mx.org`.

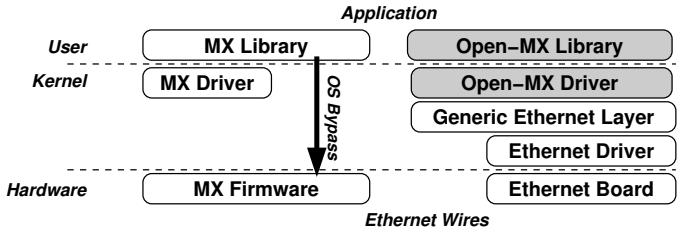


Figure 1. Design of the native MX and generic Open-MX software stacks.

hardware directly without suffering from the overhead of features that are not that useful in clusters, such as congestion control. It then exposes the *Myrinet Express* API (MX) [18] to user-space applications so that many existing middleware projects such as Open MPI [9] or PVFS2 [20] run successfully unmodified on top of it. Open-MX is also interoperable with hosts running the native MX stack over Ethernet (MXoE). This wire compatibility is a key feature of Open-MX. It is under experimentation at Argonne National Laboratory to provide a PVFS2 transport layer between BlueGene/P compute and storage nodes. The compute nodes running Open-MX are connected through a Broadcom 10-gigabit Ethernet interface to storage nodes with a Myri-10G interface running the native MXoE stack.

To achieve these goals, Open-MX was first designed as an emulated MX firmware in a Linux kernel module [10]. This way, legacy applications built for MX benefit from the same abilities without needing the Myricom hardware or the native MX software stack (see Figure 1). However, the features that are usually implemented in the hardware of high-speed networks are obviously prone to performance issues when emulated in software. Indeed, portability to any Ethernet hardware requires the use of a common very simple low-level programming interface to access drivers and NICs.

The inability of generic NICs to implement advanced mechanims such as zero-copy data transfer leads to many possible cache efficiency issues. Reducing cache effects in the Open-MX stack requires to ensure that data structures are not used concurrently by multiple cores. Since the send side is mostly driven by the application, the whole send stack is executed by the same core. The receive side is however much more complex. As any other Ethernet-based receive stack, Open-MX processes incoming packets in its *Receive handler* which is invoked when the Ethernet NIC raises an interrupt. The receive handler first acquires the descriptor of the communication channel (*endpoint*). Then, if the packet is part of a *eager* message (i.e. $\leq 32\,\text{kB}$), the data and corresponding event are written into a ring shared with the user-space library. Finally, the library will copy the data back to the application buffers (see Figure 2).

If the packet is part of a large message (after a *rendezvous*), the corresponding *Pull* handle is acquired and updated. Then, the data is copied into the associated receive buffer (Figure 2). An event is raised at the user-space level only when the last packet is received. This copy may be offloaded to Intel *I/O Acceleration Technology* (I/OAT) DMA engine hardware if available [11].

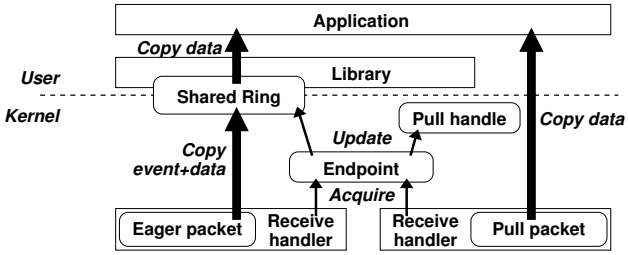


Figure 2. Resource accesses and data transfers along the Open-MX receive stack.

The current Open-MX receive stack will most of the times receive IRQ (*Interrupt ReQuest*) from all cores since the hardware chipset usually distributes them in a round-robin manner (as depicted by Figure 3(a) later). Having different cores access shared resources causes cache-lines bounces between these cores. It explains why processing all packets for the same endpoint on the same core will improve the cache-efficiency: Having a single core access the endpoint structure or shared ring in the driver leads to no more cache-line bounces and no more concurrent accesses to these shared resources. Additionally, all eager packets will also benefit from having the user-space library run on the same core since a shared ring is involved.

Large messages (*Pull* packets) will also benefit from having their handles accessed by a single core. This is actually guaranteed by the fact that each handle is used by a single endpoint. Moreover, running the application on the same core will reduce cache effects when accessing the received data (except if the copy was offloaded to the I/OAT hardware which bypasses the cache).

In the end, all incoming Open-MX packets have to be processed in the operating system on any of the cores and then passed to user-space where the application probably runs on another core. Cache efficiency thus suffers from concurrent accesses in the operating system and between the driver and the user-space application.

2.2. Related Works

Proper use of caches may have a critical impact on performance. Many research works have been carried out to improve cache efficiency of high performance computing, from cache oblivious algorithms [8] up to low-level hardware improvements. Networking communications are also subject to cache efficiency problems as explained in the previous section, but the actual issues highly depend on the hardware features and software implementation.

High-performance communication in clusters heavily relies on specific features provided by the networking hardware, such as Mellanox ConnectX [6] or Myricom Myri-10G [17]. The most famous hardware feature for HPC remains zero-copy support. It has also been added to some Ethernet-based message passing stacks, for instance by relying on RDMA-enabled hardware and drivers, such as EMP [22] or recently iWarp [21]. This strategy achieves a high

throughput for large messages. But it requires complex modifications of the operating system (since the application must be able to provide receive buffers to the NIC) and of the NIC (which decides which buffer should be used when a new packet arrives). High-speed networks do not suffer from many cache-related problems since events and data are directly deposited in the user-space application context without any intermediate cache-polluting copy. Regular ETHERNET hardware do not benefit from such a model, it only offers an interrupt-driven model. The host operating system processes incoming packets only when the NIC raises an interrupt. It then passes them to the user-space application. This mechanism prevents applications from directly polling the NIC for incoming packets. And it implies cache-line bounces unless the operating system stack and application are carefully bound to the same processor.

Several research projects specifically did target high-performance message passing over ETHERNET in the past. The most popular one is GAMMA [5] which only works on a limited hardware range since it uses a modified driver which does not support regular TCP/IP anymore. MULTIEDGE [16] uses a similar design on recent 1- and 10-gigabit hardware and thus achieves good bandwidth, but yields quite high latency levels. EMP [22] goes even further by modifying the firmware of some programmable boards to achieve better performance. Such software or hardware modifications may reduce cache-efficiency issues thanks to reduced memory copy requirements or application-directed polling. However, such implementations do not support regular hardware and software stacks. OPEN-MX relies on the generic ETHERNET layer of LINUX and thus may use any hardware. It may also coexist with the TCP/IP stack that is still often used for administration or storage purposes.

MPI/QMP [3] uses a OPEN-MX-like model, based on M-VIA, to achieve large bandwidth over multiple regular ETHERNET links. PM/ETHERNET-HXB [23] offers a similar design and supports trunked ETHERNET connections. They both achieve interesting performance levels thanks to multiple underlying ETHERNET connections, but are not designed for single high-performance connections such as MYRI-10G. OPEN-MX is designed to efficiently use modern ETHERNET hardware. It does require the aggregation of multiple links to achieve high-performance, but it may also transparently use a trunked connection to aggregate multiple links if desired. However, in the end, all these software implementations suffer from similar cache problems due to similar paths for events and data from the NIC up to the application: The operating system processes packets on different cores and then passes them to the application running on likely yet another core.

An interesting way to avoid cache-polluting memory copies is to use virtual memory tricks to remap the source buffer in the target virtual address space. Such a strategy has been studied for a long time to offer zero-copy socket implementations [4] and more recently for ETHERNET-based message passing [19]. However, even if memory copies are avoided, cache pressure remains high since remapping requires cache flushing. Also, careful binding of the operating system stack and of the application is still required so as to avoid cacheline bounces between the processing components along the receive stack. Moreover, this strategy has multiple corner-cases caused by modern operating systems heavily relying on multiple page states, pages being shared, miss-alignment, or memory pinning. It makes remapping technically difficult and expensive in many cases while this idea was indeed very interesting for performance and CPU load reduction purposes.

Some ETHERNET-specific hardware optimizations have been developed in the context of IP networks, but they were not designed for HPC. Advanced NICs now enable the offload of TCP fragmentation/re-assembly (TSO and LRO) to decrease the packet rate in the host [13]. But this work does not apply to message-based protocols such as OPEN-MX and does not improve cache efficiency. Another famous recent innovation is multiqueue support [25]. This packet filtering facility in the NIC enables interesting receive performance improvement for IP thanks to a better understanding of the location of the receive stack in the host. We look further at this idea in the following sections.

2.3. Proposal

The cache-efficiency of the receive stack is significantly related to the actual hardware and software implementation since features such as zero-copy and application-directed polling reduce cache utilization. However, all message passing stacks implemented on top of the generic ETHERNET layer such as OPEN-MX suffer from similar cache issues since packets are processed in the driver on any core (with possible concurrent accesses) and then passed to the user-space application that likely runs on another core. We propose a study of this problem in the context of OPEN-MX.

A simple way to avoid concurrent accesses in the driver is to bind the interrupt to a single core. However, the chosen core will be overloaded, causing an availability imbalance between cores. Moreover, all processes running on other cores will suffer from cache-line bounces in their shared ring since they would compete with the chosen core. In the end, this solution may only be interesting for benchmarking purposes with a single process per node (see Section 4).

As explained above, the study of cache-efficiency in the context of TCP/IP led to the emergence of hardware multiqueue support. Several modern NICs have the ability to split the incoming packet flow into several queues [25] with different interrupts. By filtering packets depending on their IP connection and binding each queue to a single core, it is possible to ensure that all packets of a connection will be processed by the same core. It prevents many cache-line bounces in the host receive stack.

We propose in this article to study the addition of OPEN-MX-aware multiqueue support. Such a feature is becoming widely available in recent 1- or 10-gigabit NICs. We expect to improve the cache-efficiency of our receive stack by guarantying that all packets going to the same endpoint are processed on the same core. To improve performance even further, we then propose to bind the target user-process to the core where the endpoint queue is processed. It will make the whole OPEN-MX receive stack much more cache-friendly. This idea goes further than existing IP implementations where the cache-efficiency property is not transferred to the application.

The intent of this work is also to demonstrate that very simple hardware features may bring interesting performance improvements. While complex hardware features have been proposed to improve networking in HPC (for instance zero-copy or application polling support), our hardware modifications are very simple and should be applicable to many legacy NICs. The OPEN-MX specific support will be *Stateless* and based on existing multiqueue support, with a new dedicated packet filtering strategy.

3. DESIGN OF A CACHE-FRIENDLY OPEN-MX RECEIVE STACK

We now detail our design and implementation of a cache-friendly receive stack in Open-MX thanks to the addition of dedicated multiqueue support in the NIC and the corresponding user process binding facility.

3.1. Open-MX-aware Multiqueue Ethernet support

Hardware multiqueue support is based on the driver allocating one MSI-X interrupt vector (similar to an IRQ line) and one ring per receive queue. Then, for each incoming packet, the NIC decides which receive queue should be used [25]. The IP traffic is dispatched into multiple queues by hashing each connection into a queue index. This idea improves performance by having multiple packets from the same connection be processed together, thus improving locality.

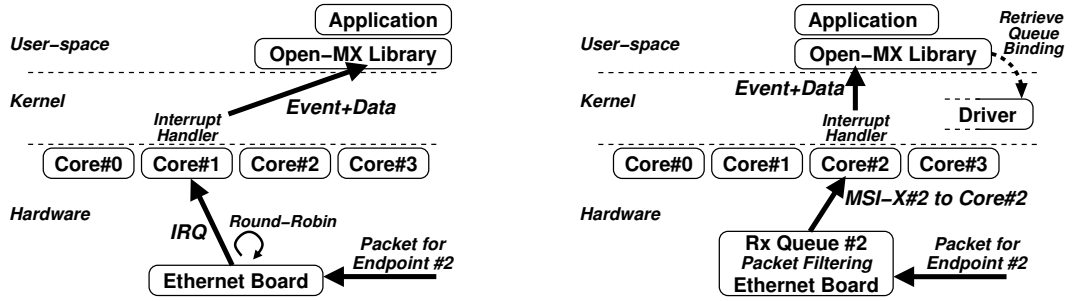
The Open-MX multiqueue support is actually very simple because hashing its packets is easy. Indeed, the same communication channel (*endpoint*) is used to communicate with many peers, so only the local endpoint identifier has to be hashed. Therefore, the NIC only has to convert the 8-bit destination endpoint identifier into a queue index. Considering the slowness of NIC processors, this conversion is much more simple than hashing IP traffic where many connection parameters (source and destination, port and address) have to be involved in the hash function. This model is summarized in Figure 3(b).

All packets for the same destination endpoint are now placed in the same receive queue, the processing of each endpoint channels may be dispatched to different cores. The next step towards a cache-friendly receive stack is to bind each process to the core which handles the receive queue of its endpoint.

3.2. Multiqueue-aware Process Binding

Now that the receive handler is guaranteed to always run on the same core for all packets of the same endpoint, we discuss how to have the application run there as well. One solution would be to move the receive queue near the target process when it actually opens the corresponding endpoint. However, moving receive queues depending on process placement may easily break their load-balance, causing multiqueueIP performance to decrease. Since Open-MX was designed to coexist with the IP stack, the binding of all queues has to remain managed globally and independently of the process placement.

We have chosen the opposite solution: keep receive queues bound as usual (one queue per core) and make Open-MX applications migrate on the right core. Therefore, when an application opens an endpoint, the Open-MX library will bind it near the corresponding receive queue as depicted on Figure 3(b) and explained in the next section. Since most high-performance computing applications place one process per core, and since most MPI implementations use a single endpoint per process, we expect each core to be used by a single endpoint. In the end, each receive queue will actually be used by a single endpoint as well. It makes the whole model very simple.



(a) Round-Robin Single-Interrupt: the interrupt goes to any core while the application may run somewhere else.

(b) Open-MX-aware Multiqueue: the NIC raises the MSI-X interrupt corresponding to the core where the application runs.

Figure 3. Path from the NIC interrupt up to the application receiving the event and data.

Additionally, this model enables the pre-warming of processor caches with incoming packets thanks to Intel *Direct Cache Access* [14]. This strategy would further improve performance by avoiding cache misses when the receive handler starts processing a new packet, but we did not have any DCA-enabled machine to test it.

3.3. Implementation

We implemented this model in the Open-MX stack with Myricom Myri-10G NICs as an experimentation hardware. We have chosen this board because it was one of the very first NICs with multiqueue receive support. It also enables comparisons with the MX stack which may run on the same hardware (with a different firmware and software stack that was designed for MPI).

We implemented the proposed modification in the `myri10ge` firmware by adding our specific packet hashing. It decodes native Open-MX packet headers to find out the destination endpoint number as specified in the MX wire specifications. Once the Ethernet driver has been setup with one receive queue per core as usual, each endpoint packet flow is sent to a single core.

Meanwhile, we added to the `myri10ge` driver a routine that returns the MSI-X interrupt vector that will be used for each Open-MX endpoint. When Open-MX attaches an interface whose driver exports such a routine, it gathers all interrupt affinities (the binding of the receive queues). Then, it provides the Open-MX user-space library with binding hints when it opens an endpoint. Applications are thus automatically migrated onto the core that will process their packets. It makes the whole stack more cache-friendly, as described on Figure 3(b).

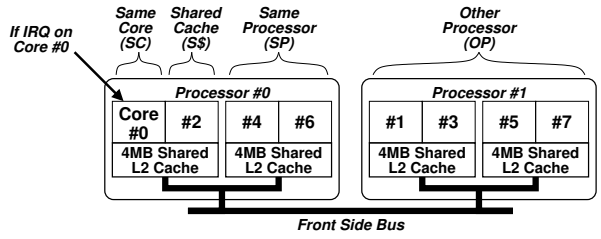


Figure 4. Processors and caches in the experimentation platform (OS-numbered).

4. PERFORMANCE EVALUATION

We now present a performance evaluation of our model. After describing our experimentation platform, we will detail micro-benchmarks and application-level performance.

4.1. Experimentation Platform

Our experimentation platform is composed of 2 machines with 2 Intel Xeon E5345 quad-core *Clovertown* processors (2.33 GHz). These processors are based on 2 dual-core sub-chips with a shared L2 cache as described in Figure 4. It implies 4 possible process/interrupt bindings : on the same core (SC), on a core sharing a cache (S$), on another core of the same processor (SP), and on another processor (OP).

These machines are connected with Myri-10G interfaces running in Ethernet mode with our modified `myri10ge` firmware and driver. We use Open MPI 1.2.6 [9] on top of Open-MX 0.9.2 with Linux kernel 2.6.26. The MPI ping-pong latency on this setup is close to 10 μs (8 μs with a native Open-MX ping-pong). It may also achieve 9 out the raw 10-gigabit/s line-rate when enabling I/OAT copy offload [11].

4.2. Impact of Binding on Micro-Benchmarks

Table I presents the latency and throughput of *Intel MPI Benchmark* [15] Pingpong depending on the process and interrupt binding. Three key results have to be noticed. First, it shows that the original model (with a single interrupt dispatched to all cores in a round-robin manner) is slower than any other model, due to cache-line bounces. Indeed, consecutive packets are never processed by the same core in the operating system. So the endpoint and pull handle descriptors keep moving from one cache to another. Additionally, the user-space application is running on a single core, so seven out of eight packets on average have to move from one cache to another when being delivered to user-space by the driver.

Secondly, when binding the single interrupt to a single core, the best performance is achieved when the process and interrupt handler share a cache but do not actually use the same core. Indeed, this case reduces the overall latency thanks to cache hits in the receive stack, while it

Table I. IMB PINGPONG performance depending on process and IRQ binding.

Metric	Binding	SC	S$	SP	OP
0 byte latency	Round-Robin Single IRQ		$\simeq 11.4\,\mu$s		
	Single IRQ on core #0	10.96	9.34	10.32	10.25
	Multiqueue		10.10		
4 MB throughput	Round-Robin Single IRQ		$\simeq 646$ MiB/s		
	Single IRQ on core #0	719	723	721	714
	Multiqueue		707		
4 MB throughput (I/OAT)	Round-Robin Single IRQ		$\simeq 905$ MiB/s		
	Single IRQ on core #0	1056	1059	1048	1026
	Multiqueue		965		

prevents the user-space library and interrupt handler from competing for the same core. This configuration is optimal when benchmarking a single process per node but obviously is not applicable to real applications with one process per core.

Thirdly, multiqueue support achieves satisfying performance, but remains a bit slower than optimally bound single interrupt. It is related to the multiqueue implementation requiring more work in the NIC than the single interrupt firmware. This overhead is actually related to the generic multiqueue support in the firmware. Our OPEN-MX specific additions only bring a dozen lines of code and two logical tests. While being a bit slower than optimally bound single interrupt, this model however works with multiple processes per node, which is what real application actually require.

4.3. Idle Core Avoidance

The above results assumed that one process was running on each core even if only two of them were actually involved in the MPI communication. This setup has the advantage of keeping all cores busy. However, it may be far from the behavior of real applications where for instance disk I/O may put some processes to sleep and cause some cores to become idle. If an interrupt is raised onto such an idle core, it will likely be asleep because of power saving, and will thus have to wakeup before processing the packet. On modern processors, this wakeup overhead is several microseconds, causing the overall latency to increase significantly.

To study this problem, we ran the previous experiment with only one communicating process per node, which means 7 out of 8 cores are idle (they were busy waiting in a MPI barrier during the previous experiment). When interrupts are not bound to the right core [†], it increases the latency from 11 up to 15-20 μs and reduces the throughput by roughly 20 %.

[†]Actually, a core only sleeps if the entire dual-core sub-chip is idle. So binding to the very next core works too.

Table II. L2 cache misses (Kernel+User) during a ping-pong.

Length	Round-Robin IRQ	IRQ on S$	IRQ on SC
0 B	29.80 % + 13.79 %	29.70 % + 8.34 %	11.47 % + **0.13 %**
128 B	26.80 % + 17.90 %	25.06 % + 23.05 %	14.15 % + **0.51 %**
32 kB	28.77 % + 17.71 %	23.17 % + 29.32 %	24.49 % + 22.56 %
1 MB (I/OAT)	27.7 % + 6.28 %	36.9 % + 7.97 %	25.20 % + 5.96 %

This result is another justification of our idea to bind the process to the core that runs its receive queue. Indeed, if a MPI application is waiting for a message, the MPI implementation will usually busy poll the network. Its core will thus not enter any sleeping state. By binding the receive queue interrupt and the application to the same core, we guarantee that this busy polling core will be the one processing the incoming packet in the driver. It will be able to process it immediately, causing the observed latency to be much lower. All other cores that may be sleeping during disk I/O will not be disturbed by packet processing for unrelated endpoints. This result may even reduce the overall power consumption of the machine.

4.4. Cache Misses

Table II presents the percentage of cache misses observed with PAPI [2] during a ping-pong depending on interrupt and process binding. Only L2 cache accesses are presented since the impact on L1 accesses appears to be lower, possibly because our overall workload is much larger than the 32 kB L1 caches.

The table first shows that the cache miss rate is dramatically reduced for small messages thanks to our multiqueue support. Running the receive handler (the kernel part of the stack) always on the same core divides cache misses in the kernel by 2. Binding the target application (the user part of the stack) to the same core further reduces user-space cache misses by a factor of up to 100.

Cache misses are not improved for 32 kB message communication. This behavior is caused by the number of copies that are involved on the receive path. Indeed, one drawback of the current Open-MX implementation up to 32 kB messages is the matching of MPI messages in user-space: it requires one copy from the kernel inside the shared ring and another copy back from the ring into the application destination buffer. These copies cause too many cache pollution, which prevents our cache efficiency improvements from being visible.

Very large messages with I/OAT copy offload do not involve any data copy in the receive path. Cache misses are thus mostly related to concurrent accesses to the endpoint and pull handles in the driver. We observe a slightly decreased cache miss rate thanks to proper binding. But the overall rate remains high, likely because it involves some code-paths outside of the Open-MX receive stack (*rendezvous* handshake in user-space, send stack, ...) which are expensive for large messages.

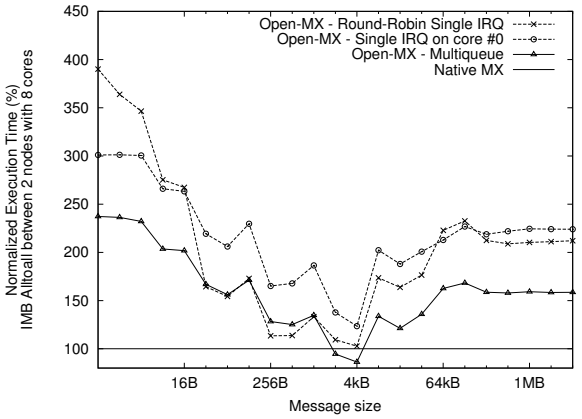


Figure 5. IMB Alltoall relative execution time depending on interrupt binding and multiqueue support, compared with the native MX stack.

4.5. Collective Communication

After demonstrating that our design improves cache-efficiency without strongly disturbing micro-benchmark performance, we now focus on complex communication patterns by first looking at collective operations. We ran IMB Alltoall between our nodes with one process per core. Figure 5 presents the execution time compared to the native MX stack, depending on interrupt and receive queue binding. It shows that using a single receive queue results in worse performance than our multiqueue support. As expected, binding this single interrupt to a single core decreases the performance as soon as the message size increases since the load on this core becomes the limiting factor.

When multiqueue support is enabled, the overall Alltoall performance is on average 1.3 better. It now reaches less than 150 % of the native MX stack execution time for very large messages when I/OAT copy offload is enabled. Moreover, our implementation is even able to outperform MX near 4 kB message sizes[‡].

This result reveals that our implementation achieves its biggest improvement when the communication pattern becomes larger and more complex (collective operation with many local processes). We think it is caused by such patterns requiring more data transfer within the host and thus making cache-efficiency more important.

[‡]Open-MX mimics MX behavior near 4 kB. This message size is a good compromise between smaller sizes (where the big Ethernet latency matters) and larger messages (where intensive memory copies may limit performance).

Table III. NAS Parallel Benchmark execution time and improvement.

	Single IRQ Round-Robin	Single IRQ on Single core	Multiqueue	Performance Improvement	MX
cg.B.16	34.62 s	34.32 s	33.68 s	+2.8 %	32.23 s
mg.B.16	4.14 s	4.19 s	4.02 s	+2.9 %	3.93 s
ft.B.16	22.80 s	23.06 s	21.34 s	+6.8 %	19.61 s
is.B.16	11.84 s	10.83 s	1.25 s	×8.5	1.33 s
is.C.16	14.69 s	14.17 s	5.62 s	×2.6	6.30 s

4.6. Application-level Performance

Table III presents the execution time of some *NAS Parallel Benchmarks* [1] between our two 8-core hosts. Most programs show a few percents performance improvement thanks to our work. This impact is limited by the fact that these applications are not highly communication intensive. IS (which performs many large message communications) shows an impressive speedup (8.5 for class B, 2.6 for class C). Thanks to our multiqueue support, IS is now even faster on Open-MX than on MX. We feel that such a huge speedup cannot be only related to the efficiency of our new implementation. It is likely also caused by poor performance of the initial single-queue model because of very poor cache efficiency Indeed, looking at cache miss rates confirms that they are dramatically reduced by our multiqueue implementation, by a factor of about 11 on IS. It is again worth noticing that using a single interrupt bound to a single core sometimes decreases performance. As explained earlier, this configuration should only be preferred for micro-benchmarking with very few processes per node.

5. CONCLUSION AND PERSPECTIVES

While HPC networking relies on complex hardware features such as zero-copy, Ethernet remains simple. The Open-MX message passing stack achieves interesting performance on top of it without benefiting from advanced features in the networking hardware. This paper presents a study of the cache-efficiency of the Open-MX receive stack.

We looked at the binding of interrupt processing in the driver and of the library in user-space. We proposed the extension of the existing IP hardware multiqueue support which assigns a single core to each connection. It prevents shared data structures from being concurrently accessed by multiple cores. Open-MX specific packet hashing has been added into the official firmware of Myri-10G boards[§] so as to associate a single receive queue with each communication channel. Secondly, we further extended the model by enabling the automatic

[§]Available in the official `myri10ge` firmware since version 1.4.33.

binding of the target end application to the same core. Therefore, there are fewer cache-line bounces between cores from the interrupt handler up to the target application.

Performance evaluations first shows that the usual single-interrupt based model may achieve very good performance when using a single task and binding it so that it shares a cache with the interrupt handler. However, as soon as multiple processes and complex communication patterns are involved, the performance of this model suffers, especially from load imbalance between the cores. Using a single-interrupt scattered to all cores in a round-robin manner distributes the load but it shows limited performance due to many cache misses.

Our proposed multiqueue implementation distributes the load as well. It also offers satisfying performance for simple benchmarks. Moreover, binding the application near its receive queue further improves the overall performance thanks to fewer cache misses occurring on the receive path and thanks to the target core being ready to process incoming packets. Communication intensive patterns reveal a large improvement since the impact of cache pollution is larger when all cores and caches are busy. Open-MX is now even able to perform faster than the native MX stack in some cases. We observe more than 30 % of improvement for Alltoall operations, while the execution time of the communication intensive NAS parallel benchmark IS is reduced by a factor of up to 8.

These results demonstrate that very simple hardware features enable significant performance improvement. Indeed, multiqueue support is becoming a standard feature that many NIC now implement. Our implementation is *Stateless* and does not require any intrusive modification of the NIC or host, contrary to usual HPC innovations. Such features that can be easily implemented in legacy NICs, open a large room for improvement of message passing over Ethernet networks.

ACKNOWLEDGEMENTS

We would like to thank Hyong-Youb Kim, Andrew J. Gallatin, and Loïc Prylli from Myricom, Inc. for helping us when modifying the `myri10ge` firmware.

REFERENCES

1. D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.
2. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.
3. Jie Chen, William Watson III, Robert Edwards, and Weizhen Mao. Message Passing for Linux Clusters with Gigabit Ethernet Mesh Connections. In *IPDPS'05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 9*, Denver, CO, 2005.
4. H. K. Jerry Chu. Zero-Copy TCP in Solaris. In *Proceedings of the USENIX Annual Technical Conference*, pages 253–264, San Diego, CA, 1996.
5. Giuseppe Ciaccio and Giovanni Chiola. GAMMA and MPI/GAMMA on gigabitethernet. In *Proceedings of 7th EuroPVM-MPI conference*, pages 129–136, Balatonfured, Hongrie, September 2000.

6. Mellanox ConnectX - 4th Generation Server & Storage Adapter Architecture. `http://mellanox.com/products/connectx_architecture.php`.
7. FCoE (Fibre Channel over Ethernet). `http://www.fcoe.com`.
8. Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-Oblivious Algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 285, New York City, NY, October 1999. IEEE Computer Society.
9. Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.
10. Brice Goglin. Design and Implementation of Open-MX: High-Performance Message Passing over generic Ethernet hardware. In *CAC 2008: Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2008*, Miami, FL, April 2008. IEEE Computer Society Press.
11. Brice Goglin. Improving Message Passing over Ethernet with I/OAT Copy Offload in Open-MX. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 223–231, Tsukuba, Japan, September 2008. IEEE Computer Society Press.
12. Brice Goglin. NIC-assisted Cache-Efficient Receive Stack for Message Passing over Ethernet. In *Proceedings of the 15th International Euro-Par Conference, Lecture Notes in Computer Science*, volume 5704 of *Lecture Notes in Computer Science*, pages 1065–1077, Delft, The Netherlands, August 2009. Springer.
13. Leonid Grossman. Large Receive Offload Implementation in Neterion 10GbE Ethernet Driver. In *Proceedings of the Linux Symposium (OLS2005)*, pages 195–200, Ottawa, Canada, July 2005.
14. Ram Huggahalli, Ravi Iyer, and Scott Tetrick. Direct Cache Access for High Bandwidth Network I/O. *SIGARCH Computer Architecture News*, 33(2):50–59, 2005.
15. Intel MPI Benchmarks. `http://www.intel.com/cd/software/products/asmo-na/eng/cluster/mpi/219847.htm`.
16. Sven Karlsson, Stavros Passas, George Kotsis2, and Angelos Bilas. MultiEdge: An Edge-based Communication Subsystem for Scalable Commodity Servers. In *Proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS'07)*, page 28, Long Beach, CA, March 2007.
17. Myricom Myri-10G. `http://www.myri.com/Myri-10G/`.
18. Myricom, Inc. *Myrinet Express (MX): A High Performance, Low-Level, Message-Passing Interface for Myrinet*, 2006. `http://www.myri.com/scs/MX/doc/mx.pdf`.
19. Stavros Passas, Kostas Magoutis, and Angelos Bilas. Towards 100 Gbit/s Ethernet: Multicore-based Parallel Communication Protocol Design. In *Proceedings of the 23rd international conference on Supercomputing (ICS'09)*, pages 214–224, Yorktown Heights, NY, June 2009. ACM/SIGARCH.
20. The Parallel Virtual File System, version 2. `http://www.pvfs.org`.
21. Mohammad J. Rashti and Ahmad Afsahi. 10-Gigabit iWARP Ethernet: Comparative Performance Analysis with Infiniband and Myrinet-10G. In *Proceedings of the International Workshop on Communication Architecture for Clusters (CAC), held in conjunction with IPDPS'07*, page 234, Long Beach, CA, March 2007.
22. Piyush Shivam, Pete Wyckoff, and D. K. Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing. In *Proceeding of Supercomputing ACM/IEEE 2001 Conference*, page 57, Denver, CO, November 2001.
23. Shinji Sumimoto, Kazuichi Ooe, Kouichi Kumon, Taisuke Boku, Mitsuhisa Sato, and Akira Ukawa. A Scalable Communication Layer for Multi-Dimensional Hyper Crossbar Network Using Multiple Gigabit Ethernet. In *ICS'06: Proceedings of the 20th International Conference on Supercomputing*, pages 107–115, Cairns, Australia, 2006.
24. Paul Willmann, Scott Rixner, and Alan L. Cox. An Evaluation of Network Stack Parallelization Strategies in Modern Operating Systems. In *Proceedings of the USENIX Technical Conference*, pages 91–96, Boston, MA, 2006.
25. Zhu Yi and Peter P. Waskiewicz. Enabling Linux Network Support of Hardware Multiqueue Devices. In *Proceedings of the Linux Symposium (OLS2007)*, pages 305–310, Ottawa, Canada, June 2007.